Program Testing and Analysis

—Mid-term Exam—

Department of Computer Science
TU Darmstadt

Winter semester 2017/18, December 22, 2017

Note: The solutions provided here may not be the only valid solutions.

Part 1 [4 points]

1. Which of the following statements is true? (Only one statement is true.)

   - [x] The goal of adaptive random testing is to spread test inputs evenly across the input domain.
   - [ ] The goal of adaptive random testing is to generate inputs by mutating existing inputs.
   - [ ] The goal of adaptive random testing is to test the program with the same value multiple times to find non-deterministic behavior.
   - [ ] The goal of adaptive random testing is to cover as many inputs as possible within a given time budget.
   - [ ] The goal of adaptive random testing is to use feedback from previous test executions to steer the generation of new tests.

2. Which of the following statements is true? (Only one statement is true.)

   - [ ] Testing is effective if it shows the absence of bugs.
   - [ ] Testing overapproximates the possible behaviors of a program.
   - [ ] Testing must continue until all bugs have been found.
   - [x] Testing is effective if it reveals bugs.
   - [ ] Testing is a waste of time because most code is correct anyway.

3. Which of the following statements is true? (Only one statement is true.)

   - [ ] The execution tree of a program with loops contains back-edges to bound the size of the tree.
   - [ ] The execution tree of a program with loops has at most 20 nodes.
   - [ ] The execution tree of a program with loops is undefined.
   - [x] The execution tree of a program with loops is infinitely deep.
   - [ ] The execution tree of a program with loops contains the loop body at most once.

4. Which of the following statements is true? (Only one statement is true.)

   - [ ] Complete statement coverage implies finding all bugs.
   - [ ] Complete statement coverage implies complete path coverage.
   - [x] Complete branch coverage implies complete statement coverage.
   - [ ] Complete DU-pair coverage implies complete statement coverage.
   - [ ] Complete DU-pair coverage implies complete path coverage.

Part 2 [14 points]

Consider the following SIMP program:

```
y := !x - 3; x := !y; while !x = 1 do skip
```

1. Give the semantics of the program as a sequence of transitions of the abstract machine for SIMP that was introduced in the lecture. For your reference, the appendix provides the transition rules (copied from Fernandez' book). You only have to give the first seven transitions. Use the following template to present your solution. We provide two lines for each configuration. The template starts with the initial configuration.

   Solution:

   $\langle$`y := !x - 3; x := !y; while !x = 1 do skip` $\circ\, nil, nil, \{x \mapsto 4, y \mapsto 5\}\rangle$

   $\rightarrow \langle$`y := !x - 3` $\circ$ `x := !y; while !x = 1 do skip` $\circ\, nil, nil, \{x \mapsto 4, y \mapsto 5\}\rangle$

   $\rightarrow \langle$`!x - 3` $\circ$ `:=` $\circ$ `x := !y; while !x = 1 do skip` $\circ\, nil, y \circ nil, \{x \mapsto 4, y \mapsto 5\}\rangle$

   $\rightarrow \langle$`!x` $\circ$ `3` $\circ$ `-` $\circ$ `:=` $\circ$ `x := !y; while !x = 1 do skip` $\circ\, nil, y \circ nil, \{x \mapsto 4, y \mapsto 5\}\rangle$

   $\rightarrow \langle$`3` $\circ$ `-` $\circ$ `:=` $\circ$ `x := !y; while !x = 1 do skip` $\circ\, nil, 4 \circ y \circ nil, \{x \mapsto 4, y \mapsto 5\}\rangle$

   $\rightarrow \langle$`-` $\circ$ `:=` $\circ$ `x := !y; while !x = 1 do skip` $\circ\, nil, 3 \circ 4 \circ y \circ nil, \{x \mapsto 4, y \mapsto 5\}\rangle$

   $\rightarrow \langle$`:=` $\circ$ `x := !y; while !x = 1 do skip` $\circ\, nil, 1 \circ y \circ nil, \{x \mapsto 4, y \mapsto 5\}\rangle$

   $\rightarrow \langle$`x := !y; while !x = 1 do skip` $\circ\, nil, nil, \{x \mapsto 4, y \mapsto 1\}\rangle$

2. Suppose you continue to execute the program. Does the program terminate successfully?
   - ☐ Yes.
   - ☒ No.

Part 3 [12 points]

This question is about extending the small-step operational semantics of SIMP to measure statement coverage. The goal is to compute which commands of a SIMP program are covered while the program is executing. To this end, you should:

- Assume that each command C is annotated with its code location loc. For example, instead of $l \ := \ E$ we write $l \ :=^{loc1} \ E$, and instead of $if \ B \ then \ C_1 \ else \ C_2$ we write $if^{loc2} \ B \ then \ C_1^{loc3} \ else \ C_2^{loc4}$.
- In the first question below, extend the configuration with a third element. Choose a data structure suitable for tracking statement coverage.
- In the second question below, extend the rules for commands to update the newly added data structure whenever necessary.

For your reference, the appendix provides the transition rules of the small-step semantics as discussed in the lecture (copied from Fernandez' book).

1. Describe how to extend the configuration.

   Solution:

   Instead of $\langle P, s\rangle$, use $\langle P, s, c\rangle$, where c is a map that assigns to each location either $True$ or $False$. Initially, c maps each location to $False$.

2. Show how to extend the following rules of the semantics by giving the extended rules:

   Solution:

$$\frac{}{\langle l :=^{loc} n, s, c\rangle \to \langle skip, s[l \mapsto n], c[loc \to True]\rangle}$$

$$\frac{}{\langle if^{loc1} \ True \ then \ C_1^{loc2} \ else \ C_2^{loc3}, s, c\rangle \to \langle C_1^{loc2}, s, c[loc1 \to True]\rangle}$$

$$\frac{}{\langle if^{loc1} \ False \ then \ C_1^{loc2} \ else \ C_2^{loc3}, s, c\rangle \to \langle C_2^{loc3}, s, c[loc1 \to True]\rangle}$$

$$\frac{}{\langle while^{loc1} \ B \ do \ C^{loc2}, s, c\rangle \to \langle if \ B \ then \ (C^{loc2}; while^{loc1} \ B \ do \ C^{loc2}) \ else \ skip, s, c[loc^{1} \to True]}$$

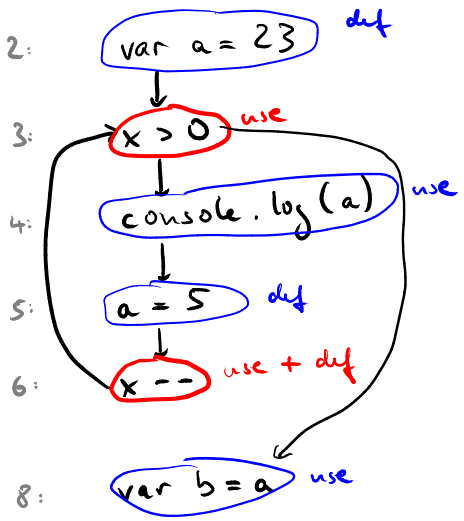
Part 4 [10 points]

Consider the following JavaScript code:

```
1 function testMe(x) {
2   var a = 23;
3   while (x > 0) {
4     console.log(a);
5     a = 5;
6     x--;
7   }
8   var b = a;
9 }
```

1. Draw the control flow graph for the above code.

   Solution:



2. In the graph that you have drawn, mark the definitions and uses of variables `a` and `x`. You do not have to consider passing the argument `x` to the function as a definition.

3. Give the DU-pairs of variables `a` and `x`. Use the line numbers of the source code to identify definitions and uses.

   - DU-pairs of variable `a`: *Solution:* (2,4), (2,8), (5,4), (5,8)

   - DU-pairs of variable `x`: *Solution:* (6,3), (6,6)

4. Suppose to test the function with the following test suite:

   - `f(1)`

   What is the DU-pairs coverage achieved by this test suite?

   Solution:

   - Covered: (2,4), (5,8), (6,3)
   - Not covered: (2,8), (5,4), (6,6)
   - 3 out of 6, i.e., 50% coverage

5. Extend the test suite to achieve 100% DU-pairs coverage.

   Solution:

   Add two test cases:

   - `f(0)`
   - `f(2)`

6. For some programs, achieving 100% DU-pairs coverage is impossible. Give an example for such a program and explain why the DU-pairs coverage cannot reach 100% for the example.

   Solution:

```
1 var x = 23;
2 var y;
3 if (false) {
4   y = x;
5 }
```

   The use of `x` at line 4 is not reachable because the condition always evaluates to false. Therefore, the DU-pair (1,4) cannot be covered.

Part 5 [10 points]

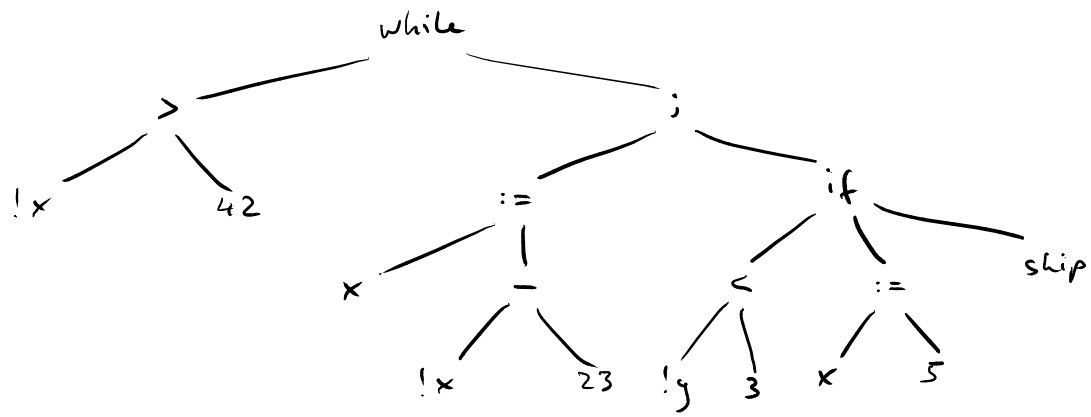
Consider the following SIMP program:

```
while !x > 42 do (x := !x - 23; if !y < 3 then x := 5 else skip)
```

1. Draw the abstract syntax tree of the program. For your reference, here is the abstract grammar of SIMP, as discussed in the lecture.

$$
\begin{aligned}
P &::= C \mid E \mid B \\
C &::= l := E \mid C; C \mid if\ B\ then\ C\ else\ C \mid while\ B\ do\ C \mid skip \\
E &::= !l \mid n \mid E\ op\ E \\
op &::= + \mid - \mid * \mid / \\
B &::= True \mid False \mid E\ bop\ E \mid \neg B \mid B \wedge B \\
bop &::= < \mid > \mid =
\end{aligned}
$$

Solution:



2. Suppose that LangFuzz uses the above program as its corpus of sample code. Give three examples of fragments of code that LangFuzz would extract from the program. For each fragment, indicate what kind of non-terminal it corresponds to.

Solution:

- B, i.e., binary expression: `!x > 42`
- E, i.e., expression: `!x`
- C, i.e., command: `x := !x - 23`

3. Suppose that LangFuzz has generated the following partial program by randomly applying grammar rules:

   `x = E; if E > E then C else C; C`

   Show two different programs that LangFuzz could create from this partial program based on a corpus of sample code that contains only the above program.

   - Program 1:

     Solution:

     `x = !x; if !x > !x then skip else skip; skip`

   - Program 2:

     Solution:

     `x = !y < 3; if !x > 5 then x := !x - 23 else x := 5; if !y < 3 then x := 5 else skip`

Part 6 [10 points]

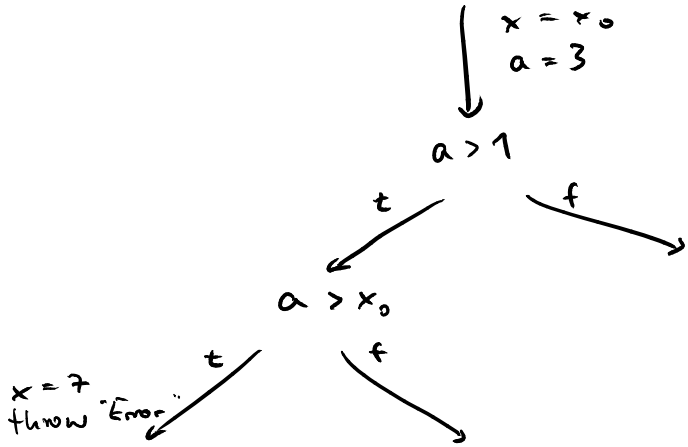
Consider the following JavaScript program:

```
1 function f(x) {
2   var a = 3;
3   if (a > 1) {
4     if (a > x) {
5       x = 7;
6       throw "Error";
7     }
8   }
9 }
```

Suppose to use concolic testing to analyze the program, where `x` is considered to be a symbolic variable.

1. Draw the execution tree of the program. If the tree is infinitely large, use "..." to represent repeating parts of the tree.

   Solution:



2. Suppose that concolic testing starts with the following concrete input `x = 5`. Illustrate the execution using the following table.

   Solution:

Line	After executing the line		
	State of concrete execution	State of symbolic execution	Path condition
2	$x = 5, a = 3$	$x = x_0$	true
3	$x = 5, a = 3$	$x = x_0$	true
4	$x = 5, a = 3$	$x = x_0$	$x_0 \geqslant 3$

3. What is the formula that concolic testing gives to the SMT solver after the first execution?

   Solution:

   $x_0 < 3$

4. Give a solution for this formula and describe what will happen if the program gets executed with the new input.

   Solution:

   $x = 2$. The program will throw an error.